Flexible Security Configuration & Deployment in Peer-to-Peer Applications

André Detsch
Luciano Paschoal Gaspary
Marinho Pilla Barcellos
Ricardo Nabinger Sanchez
Programa Interdisciplinar de Pós-Graduação em Computação Aplicada (PIPCA)
Universidade do Vale do Rio dos Sinos (UNISINOS)
São Leopoldo, Brazil

***Abstract*— The widespread adoption of P2P applications in environments beyond ordinary file sharing demands the fulfillment of several security requirements. Important steps have been taken towards security in P2P systems, with relevant mechanisms being proposed in the past to address specific vulnerabilities. However, existing approaches lack flexibility, since they do not (include enough mechanisms to) tackle a wide range of requirements in an integrated fashion. In addition, they oblige the user/application to manipulate a complex programming interface, as well as going through a cumbersome configuration process. To address these issues, we present P2PSL (P2P Security Layer), which allows gradual and flexible integration of security functionality into P2P applications. To show concept and technical feasibility, we have implemented P2PSL, assessed the overhead it induces, and incorporated the layer into a P2P-based grid computing infrastructure.**

I. Introduction

Peer-to-peer (P2P) applications have gained widespread usage in both academic and corporate environments. Although file sharing and instant messaging applications are the most traditional examples, they are no longer the only ones to profit from a P2P design. For instance, medium-sized applications, whose groups are comprised of tens or hundreds of nodes (e.g. resource sharing [1] and cooperative work [2]), are becoming increasingly common. This is also the case in the corporate arena, where P2P systems allow institutions to exchange services [3].

Although P2P applications can help provide resource sharing and collaboration in a large-scale, wide-area environments with decentralized, loosely-coupled control, their diversification and dissemination are hampered by their current lack of security. It remains difficult to develop P2P applications that address multiple combinations of security aspects, namely confidentiality, authenticity, integrity, authorization, auditing, non-repudiation, reputation, and anonymity. The reasons for that are fourfold. First, existing schemes for securing P2P applications cover only specific (security) aspects (e.g. authentication and reputation) and cannot be easily integrated into a single system. Second, they do not isolate the security aspects from the application. Instead, they oblige the user or the application developer to handle a complex programming interface and go through a cumbersome configuration process. Third, existing schemes demand a uniform, symmetric behavior of all peers that comprise an application. For some P2P applications, this limitation is undesirable, since the security requirements can vary wildly among its users. To illustrate using a trivial example, consider Skype, the VoIP application: a given peer may establish different kinds of communication with other peers, each one with its own security requirements (e.g. corporate and home users may have different needs).

Fourth and last, current schemes provide poor or no support for gradual deployment, because they need to be available in all peers of an application. It is very hard, if at all possible, to impose the abrupt adoption of a new security scheme in a large scale, loosely-coupled system. Instead, we believe that it is important for the success of P2P systems to allow the coexistence between security-enabled peers and the ones that are not.

This paper presents P2PSL, or Peer-to-Peer Security Layer, which allows the inclusion of security functionality into P2P applications and addresses the lack of *integration*, *isolation*, *asymmetry*, and *gradual deployment* just mentioned. P2PSL isolates the implementation of security aspects and their configuration from both the P2P application and the underlying communication middleware. Each peer may specify distinct security requirements (complying with different restriction degrees) for each communication channel established with other peers. In addition, the deployment of P2PSL by the peers that comprise the application can be done gradually. The implementation is based on JXTA [4], a consolidated set of protocols for P2P systems development, which has been widely used by the community.

The remainder of the paper is organized as follows. Section II discusses related work on security for peer-to-peer systems. Section III describes typical P2P applications and their security requirements in environments where security in general is desirable. Sections IV and V explain P2PSL and the peer configuration process, respectively. Section VI emphasizes implementation aspects. Section VII presents performance results obtained with the implementation, while Section VIII demonstrates the use of P2PSL through a case study in grid computing. Section IX closes the paper with concluding

remarks and perspectives for future work.

II. Related Work

The design of new schemes to simplify the development and deployment of secure P2P applications is of paramount importance to expand their use. There have been attempts to achieve that, such as JXTA and PtPTL – Peer-to-Peer Trusted Library [5]. JXTA provides functionalities like encryption, signatures, and hashes for the development of secure P2P applications. However, it obliges the programmer to explicitly include and handle security-related code, leading to additional development complexity. PtPTL, on its turn, is an OpenSSL-based API that allows for the establishment of trust between individual peer-to-peer nodes as well as the creation of secure groups of trusted peers. Both JXTA and PtPTL are restricted in relation to the security mechanisms supported: authentication, confidentiality, and integrity. When interested in employing other mechanisms such as non-repudiation, authorization, auditing, and reputation, the application developer has almost no support from the mentioned schemes.

The same limitation is observed in most (if not all) of the work on P2P security. Kim et al propose in [6] an approach to control peers joining a P2P group. The approach is comprised of two components: *Group Charter* and *Group Authority*. The former is a digital document that informs the rules for a peer to be accepted in a group while the latter is responsible for enforcing these rules. A *Group Authority* behaves as a certification authority, emitting certificates stating whether a certain member is authorized to join a given group or not. Park et al define an access control architecture for P2P applications based on RBAC (Role-based Access Control) [7]. The authors propose a generic middleware, which operates much like a service directory, indexing the resources available in the peer-to-peer network. A peer interested in providing the network with a service (or resource) is supposed to register it into the middleware. Access policies are also stored in the middleware. Whenever a peer wishes to access a service or resource, it sends a request to the middleware to find out which peer provides the service and if it is entitled to the desired service. In both papers, two important security requirements received special attention: authentication and authorization. However, like before, these are isolated implementations that cannot be easily extended or integrated to other security mechanisms.

Regarding configuration of security in P2P applications, it is particularly important to adopt more flexible, decentralized, and close-to-user security mechanisms (machine user is typically also the administrator) due mainly to scalability concerns. It is not the case of the aforementioned work ([6], [7]), which require policies to be stored and maintained in centralized servers. Approaches proposed by the web service [8] and grid computing [9] communities, in spite of their generality and completeness to secure loosely coupled distributed systems, also fail in this regard. They still depend on a great deal of configuration, which is often performed in a centralized manner.

III. P2P Applications and their Security Requirements

In this section, we consider security requirements of P2P applications. The inclusion of security functionality into applications is a hot topic in P2P research ([3], [10]). This concern is fueled due to (i) the variety of attacks P2P applications are intrinsically vulnerable to [11] (e.g. identity theft by unauthorized or falsely authorized parties, privacy invasion, loss of data integrity, and repudiation of previous transactions), and (ii) the interest of applying this technology in more important, serious activities, such as the ones related to business in the enterprise.

In Figure 1, we estimate the relevance of the main security requirements for typical P2P applications, assuming these are used in a restrictive context. Because of the relative novelty of the topic, our assessment in Figure 1 works only as a guide for the identification of security requirements.

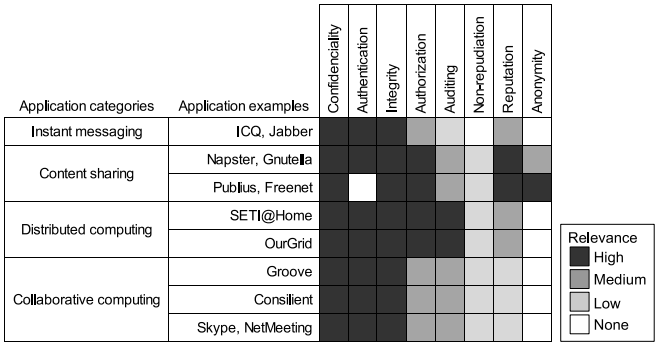


Fig. 1. Potential demand for security in typical P2P applications.

Observing Figure 1, notice that authenticity and integrity can be important to all applications enumerated. Guaranteeing confidentiality is also relevant in applications where private information is exchanged in the communication. Distributed computing applications pose more extensive security requirements, since they comprise the use of resources belonging to several administrative domains (e.g. SETI@home [12] and OurGrid [1]). Strong access control to resources and the possibility to audit system execution are particularly important in this context. Content sharing applications may have different security requirements, depending on their purpose. While applications such as Gnutella and Chord demand access control and a reputation scheme to avoid misuse (e.g. free-ride), Freenet [13] and alike aim at preserving anonymity to content provider peers. Despite the need for rigid authentication, integrity, and confidentiality, collaborative computing applications (e.g. Groove [2], Consilient [14], Skype [15]) are less demanding in relation to other security aspects. At last, instant messaging applications (e.g. ICQ [16] and Jabber [17]) have the lowest security requirements (when compared to the other application categories mentioned), but this will vary according to the nature of the remote meeting.

IV. P2P SECURITY LAYER (P2PSL)

In this section, we describe the components of the P2PSL layer. First, we discuss how security modules are combined like puzzle pieces to provide flexible, asymmetric security. Then, we describe the characterization repository, which does the mapping between modules and security requirements. Last, we show how a configuration repository is used to keep track of security requirements specified by both remote peers and the local one.

P2PSL is based on the addition of a security layer which is implemented and configured independently from both the P2P application and the underlying communication middleware. The security requirements are satisfied by modules that implement different security techniques. In line with the intrinsically decentralized nature of P2P applications, the definition and configuration of modules to be employed is done autonomously in each peer by the local user (helped by a graphical interface, as explained later). The configuration of the security layer is based on *profiles*, each one serving different security needs. So, known peers can be grouped in one of the predefined profiles; peers unknown *a priori* are automatically placed in a default profile.

Figure 2 shows an example of a network of peers using P2PSL. The configurations for Peer 2 and Peer 3 are shown in detail. To illustrate, consider Profile B in Peer 2, which specifies that authentication must be applied to all outgoing messages, and authentication and confidentiality are demanded for all incoming ones. Peer 2 then associates such profile with Peer 3, which means that Peer 2 will employ and require authentication whenever communicating with Peer 3, as well as confidentiality when receiving from it.

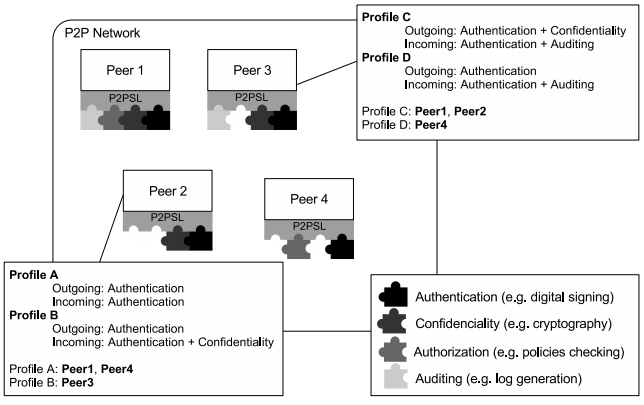


Fig. 2. Network of peers using P2PSL.

The P2PSL instance associated with the peer during system operation behaves like a wrapper between a P2P application and the underlying communication middleware. Whenever messages are received or sent, the selected modules are triggered to guarantee the security requirements established. Figure 3 illustrates this process, where Peer 2 employs Profile B to send to and receive from Peer 3. In this example, outgoing messages are passed to a digital signature module. A module may perform changes in the message according to the security requirement being enforced; this is the case in the example, where a signature is added. Incoming messages, on their turn, are passed to the digital signature and cryptography modules, also according to Profile B. If the message goes through successfully, it is delivered to the application. Otherwise, depending on the characteristics of the module, the message is simply discarded.

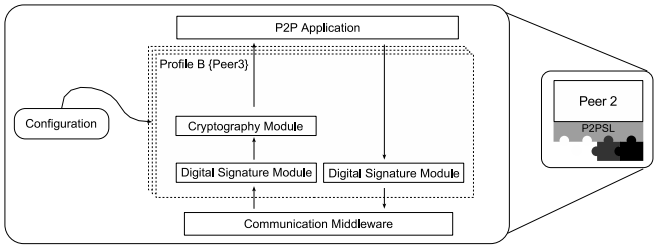


Fig. 3. P2PSL instance in a peer.

A. Security modules

The security modules are the key pieces of P2PSL. Each security technique implemented is represented through one of these modules. For example, the message auditing support is typically implemented through a log generation module; authentication and message integrity, through a digital signature module; and access control to resources (authorization), through verification of access policies.

The modules are based on the utilization of a generic interface, which allows the addition of new modules in a simple manner. Each module has methods for the verification of incoming messages and for the preparation of outgoing messages. When invoked by the security layer, the module does the processing needed (possibly changing message contents) and returns the status informing whether the operation was successful or not.

B. Characterization repository

Conceptually, the security modules differ among each other in regards to input parameters and utilization dynamics. An independent repository is used to handle such heterogeneity and avoid integrating directly into P2PSL the characteristics of each idealized module. The repository is implemented through an XML file, which contains the characterization of available modules. It defines the parameters and the usage of each module, as well as the combination of modules that fulfill each of the security requirements.

The parameters of each module must be configured so that it behaves properly. Examples of parameters are the identification of a key to be employed in ciphering or in a digital signature, the level of detail to be used in a log generation module, and the pathname of a file with access policies for an authorization module. In addition, some of the parameters specified locally can be important for other peers in the network. For example, if a peer wishes to encode a message using asymmetric cryptography, it needs to know in advance

the key employed by the destination peer. Such information is negotiated through a configuration protocol (see Section V).

P2PSL associates four basic characteristics (specified through attributes) for each module. The first attribute (`export_requirement`) specifies if message changes must be performed at the sender or not in order to allow the module to be used at the receiver. For example, in modules that involve techniques like authentication and cryptography, the outgoing message needs to be changed in order to allow the recipient to apply the technique when the message is received. On the other hand, with modules such as the log generation and the verification of access policies, the original content of the message is enough to apply the technique upon receipt. The second attribute (`obligatory_if_applied`) indicates whether or not it is mandatory to apply the module when the message is received to recover the original data. This is the case with cryptography modules, but not with authentication, as the latter only adds a signature to the message. The third and fourth attributes (`allow_on_bcast_sending` and `discard_on_failure`) regard, respectively, the possibility of employing a module in broadcast transmissions, and the need to discard a message when it fails the verification process upon receipt.

The repository stores, besides parameters and attributes of each module, the mapping between requirements and security modules. Note that this mapping is not 1:1, since a module can serve more than one security requirement, and a requirement can demand multiple modules. For instance, the combination of modules for verification via SHA-1 hash and another for signature of this verifier represents an option of message authentication. The following items map the most relevant security requirements addressed in Section III and examples of techniques that can be employed in P2PSL:

- *Confidentiality*: PGP cryptography, RSA cryptography, RC4 cryptography;
- *Authenticity*: PGP signature, RSA signature, RC4 cryptography, Message authentication code, SHA-1 hash + PGP signature on digest, SHA-1 hash + RSA signature on digest, SHA-1 hash + RC4 signature on digest;
- *Integrity*: PGP signature, RSA signature, RC4 cryptography, Message authentication code, SHA-1 hash + PGP signature on digest, SHA-1 hash + RSA signature on digest, SHA-1 hash + RC4 signature on digest;
- *Authorization*: RBAC-based access control;
- *Non-repudiation*: On-line service for evidence generation/verification, PGP signature + time-stamping, RSA signature + time-stamping;
- *Reputation*: Debit-credit reputation, feedback-based reputation, credit-only reputation.

Figure 4 shows as an example parts of a characterization file (in XML). Lines 2-22 indicate the security modules available. The definitions regarding the PgpEncryption module are shown in detail. Lines 3-7 define the usage characteristics of the module, setting each one of the four attributes previously explained. Lines 8-17 specify parameters. Lastly, lines 24-28 illustrate the mapping between the set of security requirements satisfied and the existing modules; there, it defines that confidentiality is obtained through PgpEncryption.

```
1  <static_features>
2  <modules>
3   <module name="PgpEncryption"
4           export_requirement="true"
5           obligatory_if_applied="true"
6           allow_on_bcast_sending="false"
7           discard_on_failure="false">
8    <parameters>
9     <parameter name="public_keys_file"
10             default="~/.gnupg/pubring.gpg"/>
11    <parameter name="secret_keys_file"
12             default="~/.gnupg/secring.gpg"/>
13    <parameter name="my_encryption_key_id"
14             default="" remote="true"/>
15    <parameter name="pass_phrase"
16             default=""/>
17   </parameters>
18  </module>
19  <module name="PgpSignature">...</module>
20  <module name="Log">...</module>
21  <module name="PoliciesChecking">...</module>
22 </modules>
23 <requirements>
24  <requirement name="Confidentiality">
25   <option>
26    <option_module name="PgpEncryption"/>
27   </option>
28  </requirement>
29  ...
30 </requirements>
31 </static_features>
```

Fig. 4. Example of module characterization.

C. Configuration repository

The configuration repository contains all the information required by the security layer to properly employ the modules specified by the user. The repository is implemented through an XML file and its main role is to store the configuration regarding each profile. Notice that the security requirements and modules to be applied are specified independently for incoming and outgoing communication channels. Figure 5 provides an example of profile representation; it refers to Profile B of Peer 2 shown in Figure 3. Besides the profile definition, the XML file contains the list of remote peers and their requirements, as well as standard settings for the modules locally available.

An important attribute set for each profile is named `respect_remote_requirements` (as indicated by line 2 in Figure 5). When this attribute is enabled, requirements of remote peers (i.e., security modules) are automatically satisfied (using corresponding modules) when messages are exchanged with peers belonging to the corresponding profile. Note that the set of modules applied will be the union of modules locally specified by the profile and the ones demanded by (a profile in) the remote peer.

```
<profile name="ProfileB"
         respect_remote_requirements="true">
  <incoming_requirements>
    <requirement name="Authentication"/>
    <requirement name="Confidentiality"/>
  </incoming_requirements>
  <outgoing_requirements>
    <requirement name="Authentication"/>
  </outgoing_requirements>
  <incoming_modules>
    <module name="PgpSignature"/>
    <module name="PgpEncryption"/>
  </incoming_modules>
  <outgoing_modules>
    <module name="PgpSignature"/>
  </outgoing_modules>
  <profile_peer_members>
    <profile_peer_member name="Peer3"/>
  </profile_peer_members>
</profile>
```

Fig. 5. Example of profile configuration.

V. Peer Configuration

P2P networks are expected to be dynamic, heterogeneous and asymmetric in terms of security. Because of these properties, it is unfeasible to manually configure the security layer of a peer in regards to every other peer. P2PSL tackles this problem in two ways. First, it lets users to classify peers according to profiles (as presented in the previous section). Second, the security layer includes mechanisms that guide and automate the configuration process, allowing a large number of peer relationships to be managed effortlessly.

In this section, we build on the description of P2PSL and present the three different *configuration moments* through which peers go through during their lives: (i) an initial setup that precedes the activation of a peer, (ii) a negotiation when the peer enters the network, and (iii) configuration adjustments that occur in response to P2P network changes. Each one is detailed below.

The first moment refers to the initial setup. Before a peer with P2PSL is run, it needs to create a pair of keys and publish the public one. This is required to provide authenticity and integrity in *control* messages exchanged by peers. The creation of a pair of PGP keys can be done using a tool external to P2PSL, like GnuPG or Kgpg. For publishing, there are two alternatives: one is to use PGP in a decentralized manner, whereas the other is to employ a centralized CA (Certification Authority) server. The latter is adopted in our case study. Whenever a peer receives a message signed with an unknown key, it enquires the CA about the public key of the corresponding remote peer and stores it locally for further use.

When a peer ingresses a P2P network, it needs to find about other active peers. As far as P2PSL is concerned, a peer needs to determine the set of security mechanisms demanded by each other peer it wishes to communicate with. This corresponds to the second configuration moment. The protocol that drives this communication relies on three basic communication primitives, namely *send* (unicast), *receive* and *broadcast*, to be provided by the underlying middleware, JXTA (so, the semantics for those primitives follows JXTA).

The protocol is described below through an example, which is illustrated in Figure 6. We assume, for now, there are no failures in the P2P network, and discuss later assumptions and implications about faults. Let Peer 1 be the peer that enters the network, and Peer 2 and Peer 3 peers that are already active. Assume further that Peer 2 knows about Peer 1 and has a copy of its public key, while Peer 3 and Peer 1 did not know about each other. The protocol works in three steps:

1) Peer 1, entering the network, broadcasts a signed *Requirements request* message.
2) Peer 2 receives the message, checks the signature, and immediately replies to Peer 1 with *Requirements request and reply* specifying its requirements towards Peer 1 and at the same time asking Peer 1 about its requirements towards Peer 2. Peer 3 also receives the *Requirements request* message but is unable to check it. Peer 3 fetches from the CA the public key of Peer 1, verifies the message from Peer 1, and assuming the message is correct, sends a *Requirements request and reply* to Peer 1 (similarly to Peer 2).
3) Peer 1 verifies the signature of *Requirements request and reply* received from Peer 2. Assuming the message is correct, Peer 1 sends an individual *Requirements reply* to Peer 2 containing its own requirements towards Peer 2. Peer 1 does similarly for Peer 3, but before replying it needs to fetch the public key of Peer 3 from the CA.

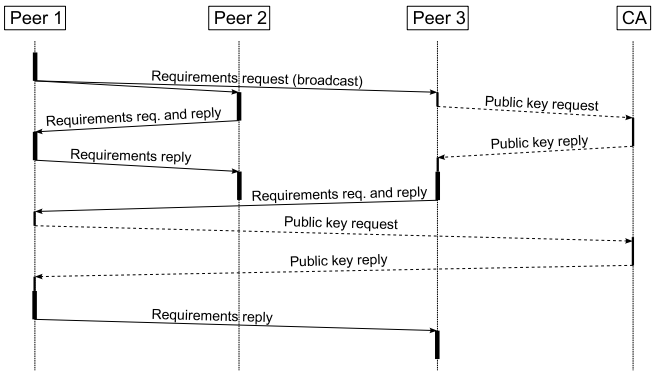


Fig. 6. Time diagram representing discovery of requirements of other peers.

Some of the messages in response to the broadcast may come from previously unseen peers. As already mentioned, the authenticity of messages is ensured by means of digital signatures. If a peer, at any moment, receives a message signed with an unknown key, it sends a request to the CA to fetch the corresponding public key. In the example, Peer 3 did not know about Peer 1 and therefore needed to fetch the public key of Peer 1 from the CA. Likewise, Peer 1 did not know about Peer 3, and consulted the CA as well. If a signed message received does not match the expected digital signature, the peer will ignore it. In addition, it places peers that (consistently)

send invalid messages in a profile associated with banned peers, and hence refuse any further communication with them.

Peers that were unknown previously are at first deemed untrusted and automatically placed in one of the two predefined profiles, as follows. The *legacy* profile refers to peers that do not have the security layer implemented or properly configured (e.g., with an invalid digital signature). The *default* profile refers to peers that implement the security layer and possess a valid digital signature, but were previously unseen by this peer (e.g., first time the peer encounters this peer id). Recall that in Figure 6 we assumed that Peer 1 did not know about Peer 3. In this case, Peer 3 is associated with the default profile.

So, at the end of the second configuration moment, a peer will have determined the set of requirements (modules to be applied locally) when sending to or receiving from every other peer it wishes to communicate with, and will have told these peers which are its own requirements.

The third and last configuration moment refers to the changes in the security configuration of a peer, which can happen at any time during its life. This is more likely in long-lived applications, where there is enough time for new peers to come in or existing peers to change their own requirements. If so, in reciprocity, a peer may wish to update its own requirements towards another peer. This would be typically achieved by moving a peer from one profile to the other. It is also possible to change the requirements associated with a profile, the one which the remote peer belongs to, but with consequences to the other peers as well.

The introduction of new requirements affects other peers and needs to be communicated to them. When a peer, say, Peer 1, changes the set of requirements towards another peer, say Peer 2, Peer 1 sends a message to Peer 2 informing the new requirements. If the set of requirements regarding Peer 2 is augmented, messages sent by Peer 1 to Peer 2 would be affected immediately. To allow a graceful increase of requirements, a transition interval can be specied by the user, temporarily delaying the application of restricting measures.

In the description above, we assumed there would be no failures in the network or in the peers. Now we consider some of the most common types of failures that can happen in a P2P system, and how P2PSL is affected by them. First, a peer may fail to receive a response to a *Requirements request* message from an active peer due to a network-related failure (partition). To handle this problem, a peer needs to employ timers to prevent indefinite waiting. A peer that fails to respond will not have its requirements registered and, therefore, will remain in the default or legacy profiles (considered unsafe for communication). So, the waiting timer needs to be long enough, since in case of peer or network contention, the arrival of messages can be arbitrarily delayed. Further, a peer may crash, in which case other peers may have to limit waiting for messages. Finally, P2PSL has no support for Byzantine failures, when a (potentially trusted) peer starts behaving arbitrarily during operation, maliciously or not.

VI. Implementation

P2PSL was implemented in Java, using JXTA ([4], [18]) as the underlying communication middleware. JXTA is a project that aims to establish a set of implementation-independent protocols that allow the creation of a general-purpose P2P structure, which can be employed by different applications. More specifically, the implementation was based on the JXTA Abstraction Layer (JAL - [19]), a library whose main goal is to ease the development of P2P applications on top of JXTA. JAL abstracts several aspects of the JXTA architecture, offering the programmer a simpler interface to access common functionality in P2P systems, like message transmission (unicast or broadcast), creation of groups or resource search. JXTA is widely used in a variety of projects and research work, allowing our implementation to be useful for several existing systems.

Figure 7 shows the P2PSL implementation overall structure. Its main piece is class `SecurePeer`, which acts as a wrapper that intercepts messages being sent or received by the peer. The modules are specializations of class `SecurityModule`, which offers a generic access interface that is employed by class `SecurePeer`. The methods in `SecurityModule` represent the verification of each incoming and outgoing message (`verifyIncomingMessage` and `adjustOutgoingMessage`, respectively) to see if they satisfy the requirements specified. The access to the modules is done solely through this generic interface. Along with dynamic class loading in module instantiation, it makes the security layer *extensible*: new modules can be added without having to recompile the rest of the layer.

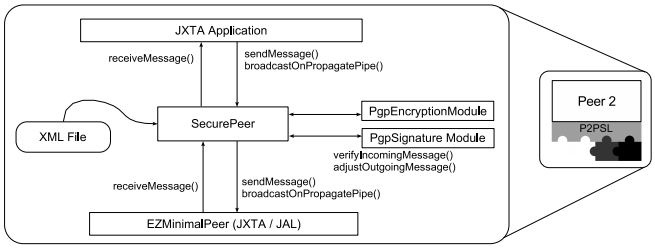


Fig. 7. P2PSL implementation.

A. Available modules

Four modules were implemented to provide the following security functionality: authentication, confidentiality, integrity, authorization and possibility to audit message exchange. New modules can be added with no change or source-code recompilation, such as when incorporating a new security mechanism or choosing an alternative technique more suitable for a given scenario. In line with P2PSL philosophy, the modules were implemented such that peers that do not have the security layer can still participate on the system. This allows a gradual adoption of the security layer in operational P2P systems, and gives the user of each peer the choice of employing or not the layer. Incidentally, peers that implement P2PSL can block

messages sent by peers that do not have it by requiring, for example, message authentication from those peers. Next, we present the modules that were implemented.

PGP signature. The PGP signature module aims to ensure authenticity and integrity of message exchanges between peers. In this model, public keys are stored in arbitrary servers, or even exchanged directly between the interested parties. The implemented module checks existing public and private keys locally available in the node where the peer runs. These keys can be managed through applications already consolidated like GnuPG and Kgpg. The private key used is established previously by the user (through an input parameter), and the generated signature is added to each message sent using a specific field. When messages are received, the signature is verified through the corresponding public key. If the signature is valid, the message can be forwarded to other modules or delivered to the application. Otherwise, and if the peer was configured to require authentication, the message is discarded. The generation of signatures is based on the BouncyCastle [20] library, which implements the PGP algorithms. Note that PGP allows the use of different signature generation algorithms, like RSA and DSA, as well as the specification of the key size. Both characteristics are specified when the pair of keys is created.

PGP cryptography. Like in the previous module, this one employs the facilities provided by PGP to guarantee message confidentiality. The PGP cryptography module available extracts from the outgoing message all fields inserted by the application, and then generates a byte array which is cryptographed and inserted in the message in a specific field. When it is received, this array is deciphered, and the fields of the original message are reconstructed and reinserted in the message, so that it remains transparent to the application. Like the signature module, routines provided by the BouncyCastle library are used.

Verification of access policies. Aiming to provide access control to resources (authorization), the module for policy verification uses generic information in the message (like date, message size and sender identification) to define if the message can be delivered or not. Specification and verification of policies are based on XACML [21], a standard created by OASIS for the definition of access control policies through XML. The policies are defined classifying the peers in roles, following the Role-based Access Control (RBAC) mechanism. Hence, two distinct repositories are established (both implemented through XML files): one for the specification of access policies, and other to fit the peers in the defined roles. Whenever a message is received, the roles played by the sender are determined and the current policies consulted, based on the information relevant to access control. This query is processed using the Sun XACML library, and returns as result whether the access can be granted. If access is not allowed, the message is silently discarded. Refer to [22] for further details on this module.

Log generation. When used, the log generation module creates, according to a preset level, a trace that presents information about the exchanged messages. Examples of information are the instant of each event, characteristics of messages and information about peers taking part in the communication. The output is directed to a file established during configuration. Through this module it is possible to audit the message exchange, identifying problems in the way the application is functioning or being used.

B. Configuration assistant

To ease the task of adjusting the configuration repository, described in Section IV-C, a Graphical User Interface (GUI) is provided. This GUI or front end is activated during the peer configuration process, described in Section V. For each profile, the user specifies the aspects to be met, and then determine the combination of available techniques to reach the desired goal.

Figure 8 presents a snapshot of the tool with the security profile configuration screen, where parameters can be specified for a module (in the case shown, a PGP signature). Once completed, the established configuration can be written to the XML file stored in the configuration repository and interpreted by the security layer.

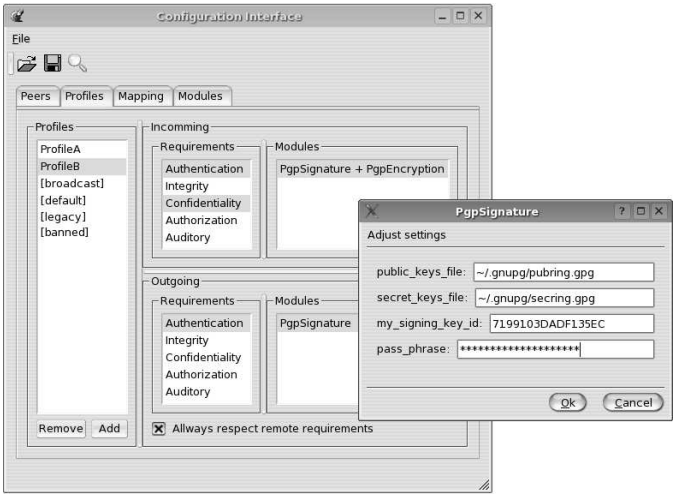


Fig. 8. GUI provided by configuration assistant.

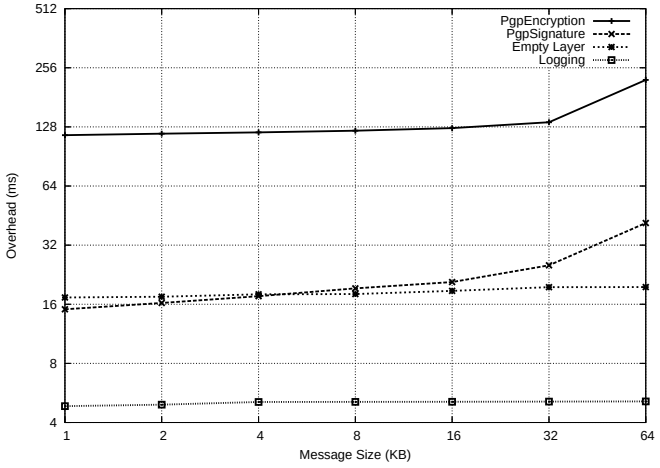
VII. Experimental Evaluation

The need for security, particularly in corporate applications, is clear. However, the required security mechanisms to be incorporated into a P2P application will introduce overheads. This section presents an experimental evaluation with the implementation of P2PSL. Our aim was to use this implementation as a proof-of-concept of P2PSL as well as make measurements of the latency overhead induced by the security layer.

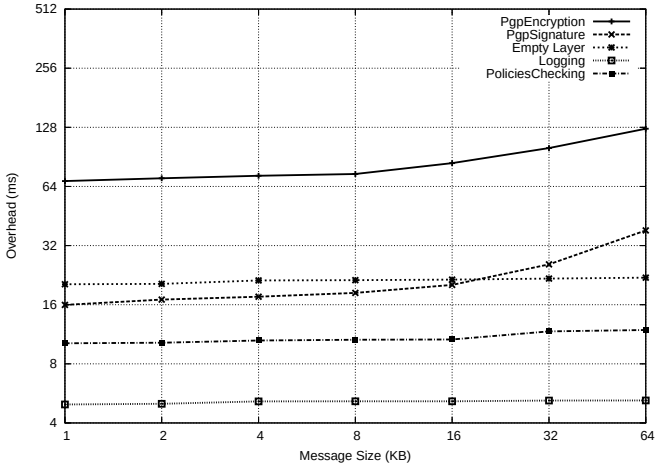
Experiments were conducted using a synthetic load in order to isolate and measure the overheads of P2PSL without modules and of each module individually. They were performed in a 2.4 GHz Intel Pentium4 CPU with 1 GB RAM. Although we

investigated different choices for algorithms and key security parameters, we report only the main results here. To obtain statistically sound results, each experiment was repeated 400 times. Adopting degree of confidence of 99%, the confidence interval seen for any experiment was no larger than 0.38 (milliseconds).

Message size is expected to play an important effect into the performance of the security layer, and so the experiments were conducted using message size as a factor: 1, 2, 4, 8, 16, 32 and 64 KiB of data, with randomly-generated contents. Figure 9 shows the average latency results for transmission (a) and reception (b) of messages. The values shown in the plots refer to P2PSL configured with no security modules (*Empty Layer*), and to each module isolatedly[1] (*PgpSignature, PgpEncryption, Logging,* and *PoliciesChecking*).



(a) send



(b) receive

Fig. 9. Overheads induced by the layer alone (no modules) and by each module individually (no layer).

Examining the figure, first notice that the empty layer takes around 20 ms in both send and receive operations, with almost no variation regarding message size. As expected, the PGP encryption module induces a very high overhead per message, whose average reaches almost 250 ms when ciphering messages of 64 KiB. The overhead will depend on the key size; in the experiments shown, we chose to use 1024 bit. The PGP signature module also induces substantial overhead, reaching 40 ms for the generation and verification of signatures in messages of 64 KiB. The module for access policy verification (receiver only) presents low overhead, between 10 and 15 ms for a single rule, but the delays will be larger according to the number of policies to be verified. The module for log generation induces a small overhead, under 6 ms, regardless of message size, because in the experiments the level of detail was set to "medium" and thus message contents were not written to the log.

Overall, individual delays were large, typically within tens or hundreds of milliseconds per message. These delays should not be considered alone, but instead combined, since a given P2P application is likely to employ multiple security modules. In the worst case scenario, if all modules were employed and messages were of 64 KiB, total delays per message would reach 491 ms, that is, around 288 ms for sending plus 203 ms for receiving.

For some applications, such delays would be unaffordable, such as in interactive P2P applications. However, this kind of application tends to use small messages (less than 1 KiB), in which case the worst-case delay would be almost halved, around 272 ms (153 ms for sending plus 119 ms for receiving). Even so, with such a delay, the maximum transmission capacity would be limited by the sender to six 1 KiB-messages per second.

Hence, the choice of modules for a specific peer will be limited not only by the existing module implementations, but also by the processing capacity available for this P2P application in the node. On the other hand, the processing delays introduce inherent costs that have been long associated with implementing security. The above study highlights the importance of flexibility and autonomy in choosing which modules a peer will employ.

VIII. Case study: P2P-based grid computing

In this section we introduce a case study, which has been carried out to show both concept and technical feasibility of our proposal. We have incorporated P2PSL into OurGrid [1], a P2P-based grid computing infrastructure.

The motivation for choosing OurGrid in this case study is twofold. First, although it has a mature software implementation and is being deployed in production environments [23], it currently lacks a robust security infrastructure, allowing various kinds of attacks or misuses of the system. Second, grid computing infrastructures demand several security requirements, as claimed in Section III, which allows us to stress the security layer.

The remainder of the section is organized as follows. First, we introduce OurGrid and present an overview of its operation

[1]not including the empty layer overhead.

protocol. Then, we describe how P2PSL has been incorporated into OurGrid and illustrate the instantiation of a secured grid infrastructure. By deploying such a setup we aimed to assess the behavior of P2PSL regarding *integration*, *isolation*, *asymmetry*, and *gradual deployment*. Finally, we assess the overhead induced by P2PSL in the execution of a real grid application.

A. OurGrid and its operation protocol

OurGrid is a P2P-based middleware that enables the creation of a multi-organization grid computing environment for the execution of bag-of-tasks applications [1]. Each organization in an OurGrid network has a *representative peer* as well as a *task scheduler*, which manages the local resources. The peer acts like a "broker" on the P2P network, trying to amass remote resources whenever local ones are insufficient to serve a request.

The main interactions of the OurGrid operation protocol are illustrated in Figure 10. When the demand for resources by users at an organization (say, Peer 1), exceeds the computing capacity locally available, its peer broadcasts a *ConsumerQuery* request message to the remaining peers (message 1 in the figure). If any organization has idle resources that satisfy the requirements of the set of tasks to be executed, the request is replied with a *ProviderWorkRequest* message (2). In the example shown in Figure 10, organization 2 replies to the request, while organization 3 does not. Upon receiving one or more replies, Peer 1 chooses the organizations where each task is going to be executed, sending them a *ProviderWorkRequestAck* message (3). For the sake of simplicity, the example shows a case where there is a single task in the set. Peer 1 then starts to exchange messages with the peer representing the chosen organization (Peer 2 in the example), in order to prepare the execution of the task (4). This phase of the protocol comprises both the creation of a temporary space at the resource to run the task and the transfer of necessary executable and data files. The task is then run (5). Once finished (6), a new phase takes place during which the resulting data is retrieved (7). Finally, after all tasks have been concluded, Peer 1 sends a broadcast message to other peers informing that the idle resources are no longer needed to satisfy the request initially sent (8).

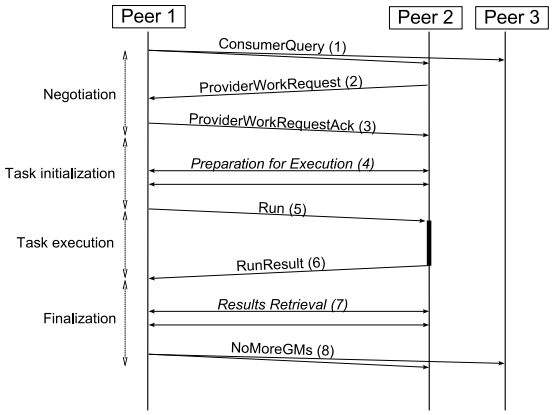


Fig. 10. Protocol used by OurGrid peers to execute tasks remotely.

B. Instantiation of a secured grid infrastructure

As OurGrid communication relies on the JXTA/JAL classes, its adaptation for P2PSL required only the replacement of the class `EZMinimalPeer` (provided by JAL) by the class `SecurePeer` (made available by P2PSL). Since `SecurePeer` extends `EZMinimalPeer`, all methods available in `EZMinimalPeer` remain available in `SecurePeer`. Therefore, no changes in both the application and the underlying communication middleware were needed, making evident the *isolation* property of P2PSL.

Having incorporated the P2PSL into OurGrid, we instantiated a real setup comprised of a dozen peers. For simplicity, we characterize a subset of the complete scenario consisting of four peers – namely Peer 1, 2, 3, and 4. Figure 11 illustrates the simplified scenario. Peer 1 represents an institution with computational tasks pending to be executed and without available resources to run them. Peers 1, 2, and 4 execute using the security layer. While Peers 1 and 2 were configured to allow mutual communication, Peer 4 was setup to block messages from Peer 1. Table I summarizes the configuration of the peers that run P2PSL. In this scenario, we ran several jobs related to bioinformatics.

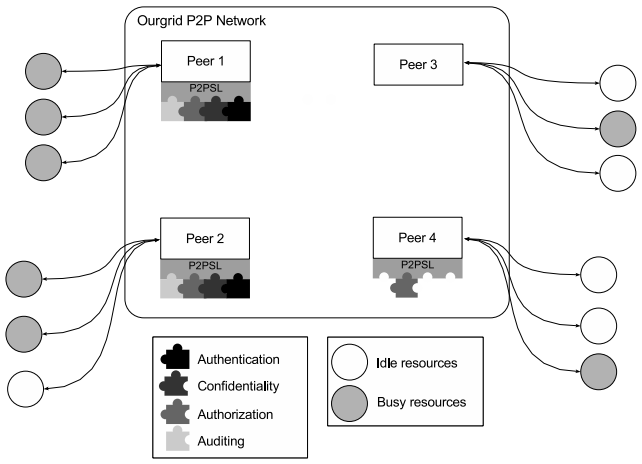


Fig. 11. Simplified view of case study scenario.

We were able to observe P2PSL enforcing policies defined by the user of a peer. For example, P2PSL running in Peer 4 discarded messages departing from Peer 1, due to a policy defined in the former. We could also experiment with the concepts of *integration*, employing several security aspects using a single system, and *asymmetry*, specifying a number of profiles for each peer and confirming that they were correctly applied in every communication channel between peers.

C. Overhead assessment

Using the same setup just described we have also assessed the communication overhead induced by P2PSL on the execution

TABLE I

CONFIGURATION OF THE PEERS RUNNING P2PSL.

Peer 1	**Profile A** *Outgoing*: Authentication (PGP signature) + Confidentiality (PGP criptography) + Auditing (log generation) *Incoming*: Authentication (PGP signature) + Confidentiality (PGP criptography) + Auditing (log generation) + Authorization (RBAC-based access control) **Profile B** *Outgoing*: Auditing (log generation) *Incoming*: Auditing (log generation) Profile A: **Peer2** Profile B: **Peer 3**, **Peer 4**
Peer 2	**Profile C** *Outgoing*: Authentication (PGP signature) + Confidentiality (PGP criptography) + Auditing (log generation) *Incoming*: Authentication (PGP signature) + Confidentiality (PGP criptography) + Auditing (log generation) + Authorization (RBAC-based access control) **Profile D** *Outgoing*: Auditing (log generation) *Incoming*: Auditing (log generation) Profile C: **Peer1** Profile D: **Peer 3**, **Peer 4**
Peer 4	**Profile E** *Outgoing*: - *Incoming*: Authorization (RBAC-based access control) Profile E: **Peer1**, **Peer 2**, **Peer 3**

of a grid application. The application used was one to compute deterministic modeling of intracellular viral kinetics ([24]). All peers ran on hosts with a 2.4 GHz Intel Pentium4 CPU, 1 GB RAM, and a Fast Ethernet network interface card. The communication profiles employed by each peer were the same enumerated in Table I.

In the communication between peers 1 and 2 the security modules were configured as follows: PGP signatures with 1024 bit DSA keys, PGP criptography with 1024 bit El Gamal keys, RBAC-based access control with a single policy, and log generation with medium level of detail (storing, for each message, sender and recipient identifiers, name of the security modules applied, message size, as well as send and receive timestamps). For the communication between Peer 1 and Peer 3 (which does not execute P2PSL), the former has been configured to employ the log generation module with high level of detail.

Table II shows the overhead (in milliseconds) induced by P2PSL on the execution of the application just mentioned. The results correspond to average values obtained by the difference between the timestamp of a message arriving at a peer and the timestamp of the next message sent by it in reaction to the message received. The values presented are (i) the computational cost of P2PSL alone and (ii) the aggregated cost of P2PSL and OurGrid to process an incoming message and react to it. These values are organized per protocol phase, namely *Negotiation*, *Task initialization*, *Task execution*, and *Finalization*, whose messages were illustrated in Figure 10.

One can observe from the table that the overhead induced by the security layer in the communication between Peer 1 and Peer 2 ranges from 35% to 55%. On the other hand, in messages exchanged between Peer 1 and Peer 3 the overhead corresponds from 8% to 11% of the total communication cost. These differences are explained by the security requirements employed in each communication channel (restrictive in the former and relaxed in the latter). The overall delays induced do not affect substantially the performance of the application running on top of OurGrid, considering the parallelism achieved in its execution and the predominance of task processing in relation to message exchange times.

IX. CONCLUDING REMARKS

The diversification and dissemination of P2P applications, specially in scenarios where extensive security requirements must be satisfied (e.g. enterprise content sharing and distributed computing), depends on the availability of flexible approaches to configure and deploy security mechanisms. As mentioned along the paper, existing approaches lack flexibility since they do not provide a wide range of requirements in an integrated fashion. Besides, they demand from the user/application the manipulation of a complex programming interface and the handling of an awkward configuration process.

To address the aforementioned issues we have proposed P2PSL (P2P Security Layer). It allows the inclusion of security functionality into P2P applications, respecting the issues of: (i) integration of security aspects into a single application; (ii) isolation between the security mechanisms and both the application and underlying middleware; (iii) asymmetry of security allowing each peer to choose, independently from each other, which requirements should be respected; and (iv) gradual deployment of the scheme in the P2P network. P2PSL has been successfully used in a P2P-based grid computing infrastructure [1].

In the future we expect to perform the incorporation of P2PSL into additional peer-to-peer applications. This will enable us to better evaluate the security layer developed, specially its generality and adherence to other applications. We also intend to develop extra security modules comprising additional security requirements, broadening the applicability of P2PSL.

REFERENCES

[1] N. Andrade, W. Cirne, F. V. Brasileiro, and P. Roisenberg, "OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing," in *Job Scheduling Strategies for Parallel Processing, 9th International, Workshop, JSSPP 2003*, pp. 61–86, June 2003.

[2] "Groove Virtual Office," Aug. 2005. http://www.groove.net/.

[3] G. Lawton, "Is Peer-to-Peer Secure Enough for Corporate Use?," *IEEE Computer*, vol. 37, pp. 22–25, Jan. 2004.

[4] L. Gong, "JXTA: A Network Programming Environment," *IEEE Internet Computing*, vol. 5, pp. 88–95, June 2001.

[5] "The Peer-to-Peer Trusted Library," Aug. 2005. http://sourceforge.net/projects/ptptl.

[6] Y. Kim, D. Mazzocchi, and G. Tsudik, "Admission Control in Peer Groups," in *Second IEEE International Symposium on Network Computing and Applications*, p. 131, Apr. 2003.

TABLE II

COMMUNICATION OVERHEAD INTRODUCED BY P2PSL IN EACH PROTOCOL PHASE, IN MILLISECONDS.

	Peer 1 - Peer 2		Peer 2 - Peer 1		Peer 1 - Peer 3		Peer 3 - Peer 1	
	P2PSL	Total	P2PSL	Total	P2PSL	Total	P2PSL	Total
Negotiation	653	1182	753	1401	84	765	-	1590
Task Initialization	502	991	555	1467	104	983	-	684
Task Execution	-	-	625	182864	-	-	-	222287
Finalization	685	1574	455	2721	47	568	-	2324

[7] J. S. Park and J. Hwang, "Role-based Access Control for Collaborative Enterprise In Peer-to-Peer Computing Environments," in *Proceedings of the eighth ACM symposium on Access control models and technologies*, pp. 93–99, 2003.

[8] "Web Services Security (WS-Security) Specification," June 2004. http://www-106.ibm.com/developerworks/webservices/library/ws-secure.

[9] V. Welch and et al., "Security for Grid Services," *IEEE Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.

[10] N. Daswani, H. Garcia-Molina, and B. Yang, "Open Problems in Data-sharing Peer-to-Peer Systems," in *ICDT 2003*, pp. 1–15, Jan. 2003.

[11] F. DePaoli and L. Mariani, "Dependability in Peer-to-Peer Systems," *IEEE Internet Computing*, pp. 54–61, July 2004.

[12] "SETI@home project," Aug. 2005. http://setiathome.ssl.berkeley.edu/.

[13] I. Clarke and S. Miller, "Protecting Freedom of Information Online with Freenet," *IEEE Internet Computing*, vol. 6, pp. 40–49, Feb. 2002.

[14] "Consilient," Aug. 2005. http://www.consilient.com/.

[15] "Skype," Aug. 2005. http://www.skype.com/.

[16] "ICQ," Aug. 2005. http://www.icq.com/.

[17] "Jabber Software Foundation," Aug. 2005. http://www.jabber.org/.

[18] S. R. Waterhouse, D. M. Doolin, G. Kan, and Y. Faybishenko, "JXTA Search: A Distributed Search Framework for Peer-to-Peer Networks," *IEEE Internet Computing*, vol. 6, pp. 68–73, Feb. 2002.

[19] "JAL - JXTA Abstraction Layer," Aug. 2005. http://ezel.jxta.org/jal.html.

[20] "Bouncy Castle Project," Aug. 2005. http://www.bouncycastle.org/.

[21] S. G. et al., "eXtensible Access Control Markup Language (XACML) Version 1.1. Committe Specification," Aug. 2003.

[22] J. F. da Silva, L. P. Gaspary, A. M. P. Barcellos, and A. Detsch, "Policy-based Access Control in Peer-to-Peer Grid Systems," in *6th IEEE/ACM International Workshop on Grid Computing*, Nov. 2005 (to appear).

[23] "Ourgrid Project," Aug. 2005. http://www.ourgrid.org/.

[24] R. Srivastava and L. and J. Yin, "Stochastic vs. Deteministic Modeling of Intracellular Viral Kinetics," *Theory Biology*, vol. 218, pp. 309–321, 2002.